MERN:

jQuery: Js library/ simple code for interactivity

React: Dev by FB

Library than a framework

Can plug react in another framework and work still

Framework dictate structure

Componential/modular

Written on JS and JSX

Third party router needed

Angular: framework

Made with typescript and HTML

Node.JS:

Powerful and flexible js runner

Features of JS:

1. Interpreted language: parse->machine code
2. Event-driven lang
3. Run on single thread: runs funct by funct strictly

ECMAScript and ES6:

-ECMA: standard of JS

- JS- implementation of ECMA standard

Syntax Errors: unexpected token: something not expected

Syntax error: invalid JS

Runtime error: bugs

Scope: visibility of information (variables/functions) to a particular section of code

Global local

Declaring variables: const let -> scope information more precisely

Block scoping – code block = {}

Let – allows for reassignment of declared variables constant

-does not require a value upon declaration

Const- must assign a value at creation

Immutable

Destructuring: JS expression that unpacks each unique variable

Rest/Spread: rest of dataset/ spread content of existing structure

Spread: makes array with rest of the existing structure

Arrow functions:

Anonymous function: function without a name

JS not good for OOP

String number Boolean

Mongoose= mongo’s middleware method to query

Mongo: DB

Express: middle

React: Frontend

Node: backend

React: JS library for building UI

Frontend

Build single page application

Feel more responsive by eliminating page loads for users

Can choose libraries we use

Doesn’t prevent DOM manipulation

Popular

Using React: uses objects called components – breaks down DOM with variety of components that have specific behaviors

Need react scripts in HTML

JSX and Babel:

Use babel to make JSX work – bring back HTML code reduce code written

Import {React} from ‘react’

Folder Structure:

Npx create-react-app name-name-name

Npm run start

Entry point- index.js call reactdom.render send in components

Rendering all of content in app.js

Put all react code in component

Components: div that contains all styling and JS/ must have a name starting with capital letter

1. Function that returns a React Element – JSX
2. Import React from react
3. Have a name starting with capital letter
4. Returns a single parent react element using JSX
5. Exports component name

Functional V Class components:

React -> components as classes or functions

Hook- state on functional component

Properties(props) – empty JS object that is constructed and passed to each react component by default

Props passed one direction / Parent Child relationship:

Parent <- component single parent element ->child component

Props cannot change

Synthetic Events:

Custom approach to adding event listeners

Wrapping browsers native elements and applying its own event system called synthetic events

1; camel cased

2; returning false does not work -> event.stopPropogation()

3; Events cannot be called asynch – event.preventDefault()

How react pools the synthetic events

onChange(e.target.value)

onSubmit

onFocus

onBlur

Hooks:

Function built in or custom that allows hook in or use certain piece of react funct inside your functional components

Not persist data -> hooks to maintain state

Current state value = state changes if via setState

Usestate useeffect

Holds it behind so not lost in re-render

Getter: describe the data that it is holding

Setter: set getter /asynch

Const [getter,setter]

Forms:

2 ways to deal with forms in react

State: track input values as piece of state – controlled components

1. Updating them on change events
2. Rendering back

Refs:

1. Attach a pointer DOM nodes like inputs and text areas and reach out to inspect their values once our form is submitted – uncontrolled components

Map and Filter:

-avoiding writing loops

Map:

Copy of an array – call back function that takes in each element of the array – returning what I want to do with each element

Filter:

If function true = append to array and vice versa

Conditional Rendering:

-render sth in the DOM that is based off on some other value

- we want output after form was submitted

1. Using a function to return the text that you would like to display
2. Ternary operators: can replace form message JSX can use to validate

Lifting State:

-need to share data between multiple components?

-lift state up to common parent pass down state getter setter as needed

Generally, app.js

Props can include functions

API’s:

Promise: need indef time to get data back without locking app

Pattern where eventual response is not known

3 states- pending resolved rejected

Better control of asynch flow

Easy to read

Avoiding callback hell

Intro to rest API’s

API that communicates among and between web services

Callback functions:

A function that is passed into another function to be called by that function

Often used with anonymous functions

Use when: - some code needs to fun after an event

Making API calls

Querying a DB

useEffect: - keeps on rendering

-manage side effects in our react project

- data fetching etc.

- useEffect in functional component -> execute directly after the component is rendered and every time the component updates

Common way to use API calls

Takes optional second argument -> more specific to tell what exactly we want it to run

Variable changes – run in the array

Use effect on every first render

When changes to diff route= cleanup so no memory leak

Axios- library that allow us to easily configure api calls in js and react app

What is Routing?:

Single Page App: not actually making another request-> pseudo-route-> tell SPA what part of app we want to see

Anatomy of URL:

<HTTP://learn.codingdojo.com/m/124/234234/23523525>

Protocol subdomain domain topleveldomain path

Routes on server side and client side

No request to server on SPA

Routes to manipulate DOM to swap out components

React Router:

-BrowserRouter: wrap around all components

-useNavigate: redirect only use in one route

useParams: access information from our routes

Express: framework written in JS which acts as an interface to Node’s server functionality – allows us to create a server

Postman: dev tool that allows us to interact with restful API’s made by others and or ourselves

HTML requests without writing a lot of code

HTTP Methods:

GET: used for passing insensitive information

Query parameters are sent as a part of the URL

Get requests can be cached and will even remain stored in your browser’s history

GET requests can be bookmarked

POST: passing sensitive information

Inside http request body – never cached – can’t be bookmarked

Forms that control user reg user auth user authori and creating a new object to store in the DB

MIDDLEWARE

USE POSTMAN TO TEST IF WE DON’T HAVE FRONTEND

PUT: passing updating whole sensitive information

Same as POST req but used to update whole entity in database

PATCH: passing updating pieces of sensitive information

Same as put / post but for updating

DELETE: passing deleting sensitive information

Delete requests are often used to delete an entity from the DB

Native HTML forms w/o JS only GET and POST but many web frameworks could fake HTTP verb with a hidden input

EXPRESS SERVER:

To start, we need to create a new project and create a new server.js file inside our newly created project folder **(if you downloaded the project, this is done for you already).**

In our **server.js** file we need to import the express module using JavaScript's require()statement, and then invoke express.

*const* express = require("express");

*const* app = express();

*const* port = 8000;

Last but not least is the app.listen() line of code that actually runs our server on a specified port.

// this needs to be below the other code blocks

app.listen( port, () => console.log(`Listening on port: ${port}`) );

node server.js = running the server

run server using n s.js – have to restart server manually every time we update code – nodemon npm install -g nodemon sudo npm install -g nodemon

AND THEN nodemon server.js TO RUN SERVER

ROUTING:

Because we are using Express primarily to build our APIs, it's best practice to start every such route with "/api" which will help us avoid route collisions with React's client-side routing.

GET:

POST: To be able to access POST data, we need to be able to pull it out of the request object. To do this, we first must add a new setting to our server.js file:

NEED THIS MIDDLEWARE

// make sure these lines are above any app.get or app.post code blocks

app.use( express.json() );

app.use( express.urlencoded({ extended: true }) );

copy

both express.urlencoded() and express.json() are ***Express middleware functions***. They are responsible for providing and parsing the request.body data.

Now that we have included our middleware, here's how we get form data:

app.post("/api/users", (req, res) *=>* {

    // req.body will contain the form data from Postman or from React

    console.log(req.body);

    // we can push it into the users array for now...

// later on this will be inserted into a database

    users.push(req.body);

    // we always need to respond with something

    res.json( { status: "ok" } );

});

Route Parameters:

Any data I wish to pass VIA URL indicated by “:”

 It will then be available in the req.params object:

// if we want to get a user with a specific id, we can make the id a part of the url

// be sure to preface the id variable with a `:` colon

app.get("/api/users/:id", (req, res) *=>* {

    // we can get this `id` variable from req.params

    console.log(req.params.id);

    // assuming this id is the index of the users array we could return one user this way

    res.json( users[req.params.id] );

});

Update Data

updating data using a put request:

app.put("/api/users/:id", (req, res) *=>* {

    // we can get this `id` variable from req.params

*const* id = req.params.id;

    // assuming this id is the index of the users array we can replace the user like so

    users[id] = req.body;

    // we always need to respond with something

    res.json( { status: "ok" } );

}); copy

Deleting Data

deleting data using a delete request.

app.delete("/api/users/:id", (req, res) *=>* {

// we can get this `id` variable from req.params

*const* id = req.params.id;

// assuming this id is the index of the users array we can remove the user like so

users.splice(id, 1);

// we always need to respond with something

res.json( { status: "ok" } );

});

copy

FOLDER STRUCTURE:

Graphical user interface, application

Description automatically generated

* server - will hold all server related files
  + config - will handle the database configuration and connection
  + controllers - will hold all logic for each model (i.e creating, updating, etc...)
  + models - will hold all the schemas
  + routes - will handle all of our routes for each model
  + server.js - will handle all the server logic with express

# **Modules**

The ability to include code from other files within another file is extremely important in a back-end environment. If you recall, we do this with front-end JavaScript by adding script tags with the src attribute pointing to the right place. But in NodeJS, we need to be able to pull code from JavaScript files into other JavaScript files so a simple script tag will not do any good for us.

To do this, we can use JavaScript’s built-in tools to import and export between files:

1. require("./filePathGoesHere") - used to import from different files
2. module.exports = WhateverYouNeedToExportGoesHere - used to export whatever content you need to export from that file

### Exporting in Node

You can download the boilerplate project [HERE](https://s3.amazonaws.com/General_V88/boomyeah2015/codingdojo/curriculum/content/chapter/Modules.zip) so you can follow along.

There are multiple ways we can export code using module.exports.

Here are a couple of ways we can export multiple functions using an object:

module.exports.greet = function() {

console.log("Hello! We are exporting a function called Greet.");

};

module.exports.add = function(num1, num2) {

console.log("The sum is: ", num1 + num2);

};copy

module.exports = {

greet: function() {

console.log("Hello! We are exporting a function called Greet.");

},

add: function(num1, num2) {

console.log("The sum is: ", num1 + num2);

}

}copy

Both export methods are exporting the same function just in a different way. The syntax in which you choose to export is your choice.

### Importing in Node

There are also multiple ways we can Import our functions:

Save our exports in a variable:

*const* myCustomModule = require("./my\_module");

myCustomModule.greet();

myCustomModule.add(5, 7);copy

Or we can use destructuring:

*const* { greet, add } = require("./my\_module");

greet();

add(5, 7);copy

### IMPORTANT NOTES:

You'll notice that we require the string **"./my\_module"**. There are 2 things to note here:

1. There is no .js at the end of the file.The require method automatically looks for JavaScript files, so we don't need to include the .js file extension.
2. The files **app.js** and **my\_module.js** are in the same directory. Normally, the **require()**method looks for node modules that aren't in the same directory as the file that is running; by default, the **require() method looks for the modules located in a folder called node\_modules.** To tell **require() to look in the current directory** (i.e. the folder that the file is located in) **we have to include "./" in front of the file path. "./" (dot-slash) is the file path for the current directory**

MONGODB / MONGOOSE

-store JS objects directly

-No SQL database

Not only sql db – supports sql but more flexibility in the storage of the data than tabular storage

* NO JOINS in nosql

| Database Type: | **SQL** | **Mongo** |
| --- | --- | --- |
| **Database** | Schema | Database (db) |
| **Collection of related records** | Tables | Collections |
| **Each one record in the collection of records** | Row / Record | Document |

**Operators**are an important part of MongoDB.

* server - This is your backend server / project folder and will hold all server related files
  + config - will handle the database configuration and connection
  + controllers - will hold all logic for each model (i.e creating, updating, etc...)
  + models - will hold all the schemas
  + routes - will handle all of our routes for each model
  + server.js - will handle all the server logic with express
* npm init -y
* npm install mongoose express

const mongoose = require('mongoose');

*const* mongoose = require('mongoose');

mongoose.connect('mongodb://127.0.0.1:27017/name\_of\_your\_DB', {

useNewUrlParser: true,

useUnifiedTopology: true

})

.then(() *=>* console.log('Established a connection to the database'))

.catch(err *=>* console.log('Something went wrong when connecting to the database ', err));

const mongoose = require('mongoose');

const UserSchema = new mongoose.Schema({

name: {

type: String

},

age: {

type: Number

}

});

const User = mongoose.model('User', UserSchema);

module.exports = User;

IN THE CONTROLLERS

const User = require('../models/user.model');

module.exports.findAllUsers = (req, res) => {

User.find()

.then((allDaUsers) => {

res.json({ users: allDaUsers })

})

.catch((err) => {

res.json({ message: 'Something went wrong', error: err })

});

}

module.exports.findOneSingleUser = (req, res) => {

User.findOne({ \_id: req.params.id })

.then(oneSingleUser => {

res.json({ user: oneSingleUser })

})

.catch((err) => {

res.json({ message: 'Something went wrong', error: err })

});}

module.exports.createNewUser = (req, res) => {

User.create(req.body)

.then(newlyCreatedUser => {

res.json({ user: newlyCreatedUser })

})

.catch((err) => {

res.json({ message: 'Something went wrong', error: err })

});}

module.exports.updateExistingUser = (req, res) => {

User.findOneAndUpdate(

{ \_id: req.params.id },

req.body,

{ new: true, runValidators: true }

)

.then(updatedUser => {

res.json({ user: updatedUser })

})

.catch((err) => {

res.json({ message: 'Something went wrong', error: err })

});}

module.exports.deleteAnExistingUser = (req, res) => {

User.deleteOne({ \_id: req.params.id })

.then(result => {

res.json({ result: result })

})

.catch((err) => {

res.json({ message: 'Something went wrong', error: err })

});}

ROUTES: ORDER MATTERS

const UserController = require('../controllers/user.controller');

module.exports = app => {

app.get('/api/users', UserController.findAllUsers);

app.get('/api/users/:id', UserController.findOneSingleUser);

app.put('/api/users/:id', UserController.updateExistingUser);

app.post('/api/users', UserController.createNewUser);

app.delete('/api/users/:id', UserController.deleteAnExistingUser);

}

SERVER

const express = require("express");

const app = express();

require("./config/mongoose.config");

app.use(express.json(), express.urlencoded({ extended: true }));

const AllMyUserRoutes = require("./routes/user.routes");

AllMyUserRoutes(app);

app.listen(8000, () => console.log("The server is all fired up on port 8000"));

npm install cors

Diagram

Description automatically generated